XGBoost: A Scalable Tree Boosting System

> *XGBoost: A Scalable Tree Boosting System*
> T.Chen, C.Guestrin
> ACM SIGKDD, 2016

박이현 주혜인 홍현경

ABSTRACT

Tree boosting은 다방면에서 효율적으로 활용되는 머신러닝 방법이다. 이 논문에서는 **XGBoost**라는확장가능한 종단간(end-to-end) Tree boosting system을 소개한다. 또한 Sparse data를 위한 sparity-aware algorithm, 근사 트리 학습을 위한 weighted quantile sketch를 제안한다. 더욱 중요하게, cache accesss 패턴, data compression 과 sharding에 대한 인사이트를 확장가능한 tree boosting 시스템을 위해 제공한다. 이러한 인사이트들을 결합함으로써 **XGBoost**는 현존하는 시스템들보다 훨씬 적은 자원을 사용하여 수많은 예제들을 해결하며 확장한다.

1. INTRODUCTION

machine learning 과 data-driven approaches는 많은 분야에서 성공적으로 활용되고 있는데, 이를 가능하게 하는 두 요인이 있다.

- 복잡한 데이터 의존성을 포착하는 효율적인 (통계적) 모델의 활용
- 대용량의 데이터셋을 이용하여 모델을 학습하는 확장가능한 학습 시스템

실제 활용되는 머신러닝 방법중에서 gradient tree boosting은 특히 많은 분야에서 활용된다.

Gradient tree boosting

gradient boosting이란, 약한 분류기를 결합하여 강한 분류기를 생성하는 것을 의미한다.

- gradient

  gradient는 쉽게 말하면 기울기로 어떤 함수 f가 다중변수 $x_1, x_2, x_3, ...$으로 구성되어 있을 때 f의 그라디언트는 각 변수에 대하여 f를 편미분 한 것을 행으로 나열한 것이다.

  어떤 스칼라함수 f에 대해서 그라디언트 ∇f의 방향이 f가 가장 빨리 증가하는 방향을 가리킨다.

- residual fitting 과 gradient의 관계

  residual : loss fn을 squared error로 설정하였을 때 negative gradient는 residual이 된다.

  ⇒ residual에 fitting하여 다음 모델을 순차적으로 만들어 나가는 것은 negative gradient를 이용해 다음 모델을 만들어 나가는 것으로 볼 수 있다.

$$j(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$$

$$\frac{\partial j(y_i, f(x_i))}{\partial f(x_i)} = \frac{\frac{1}{2}(y_i - f(x_i))^2}{\partial f(x_i)} = f(x_i) - y_i$$

$$residual = y_i - f(y_i)$$

  따라서 gradient boosting은 다음 모델을 만들때, negative gradient를 이용해 만들기 때문에 gradient boosting이라고 할 수 있다.

  residual fitting model은 gradient boosting의 한 종류이며 다른 loss function을 활용하여 gradient boosting이 가능하다.

이 논문에서는 tree boosting을 위한 확장가능한 머신러닝 시스템인 XGBoost에 대해 설명한다. 오픈소스 패키지로서 이용이 가능하며 많은 대회에서 활용되고 있다.

- scalability

  XGBoost의 가장 큰 성공요인은 scalability이다.

  > scalability란, 확장성 또는 범위성으로 해석이 가능한데 가용한 자원에 따라 성능이 크게 떨어지지 않으면서 시스템에 맞에 돌아갈 수 있는 능력을 의미한다.
  >
  > ML분야에서 알고리즘상 더 효율적인 구조를 찾거나 원본 알고리즘에 근사한 성능을 내면서 훨씬 효율적으로 계산이 가능한 방법을 찾는다는 의미다.
  >
  > 즉, data의 양과 상관없이 사용가능하면서 계산 자원을 과도하게 요구하지 않는 것을 의미한다.

  모든 시나리오에서 scalable하며 하나의 machine에서도 기존의 유명한 방법보다 10배 이상 빠르고, 분산되거나 메모리가 제한된 환경까지 확장할 수 있다. XGBoost의 확장성은 여러가지 중요한 시스템과 알고리즘적 최적화 같은 아래의 혁신 때문이다.

  - sparse data를 다루기 위한 새로운 트리 학습 알고리즘
  - 이론적으로 증명된 weighted quantile sketch 과정이 근사 트리 학습에서 instance weight를 다룰 수 있게 한다.
  - 병렬과 분산 컴퓨팅은 학습을 빠르게 하여 더욱 빠른 모델 탐색을 가능하게 한다.

out-of-core computation을 사용하고 데스크탑에서 수많은 예제들을 처리할 수 있게 한다. 또한 이러한 기술들을 결합하여 최소한의 클러스터 자원을 사용해 더 큰 데이터로 확장 가능한 end-to-end 시스템을 만들 수 있다.

종단간 학습 end-to-end learning

종단간은 처음부터 끝까지라는 의미로, 입력에서 출력까지 한번에 처리하는 것을 의미한다.

> 밑바닥부터 시작하는 딥러닝 (p.110)

이 논문의 주요한 contribution은 다음과 같다.

- highly scalable한 종단간 트리 부스팅시스템을 설계하고 구축한다.
- 이론적으로 증명된 효율적 계산을 위한 weighted quantile sketch를 제안한다.
- 병렬 트리 학습을 위한 새로운 sparsity-aware algorithm을 제안한다.
- out-of-core 트리 학습을 위해 효율적인 cache-aware block structure를 제안한다.

또한 regularized learning objective를 제안한다.

2. TREE BOOSTING IN A NUTSHELL

- **XGBoost**에 대한 내용을 이해하려면 먼저 기본이 되는 **Tree Boosting**에 대한 개념이 선행되어야 한다.
- 해당 부분에서는 **Tree Boosting**에 대해 간단히 짚어보도록 한다.

2.1 Regularized Learning Objective

$$\mathcal{D} = \{(x_i, y_i)\} \ (|\mathcal{D}| = n, \mathbf{x_i} \in \mathbb{R}^m, y_i \in \mathbb{R})$$

- 주어진 데이터셋이 n개의 obs, m개의 변수로 이루어졌을 때, 데이터셋 $\mathcal{D}$는 다음과 같이 정의할 수 있다.
- 해당 데이터셋에 대한 예측을 위해 트리 앙상블 모델은 K개의 **additive function**을 사용한다.
  - **참고 : Boosting과 Additive function**
    - Boosting은 단순한 weak learner를 이용해 strong learner를 만드는 과정인데, 이 과정에서 사용되는 함수를 **Additive function**이라고 한다.

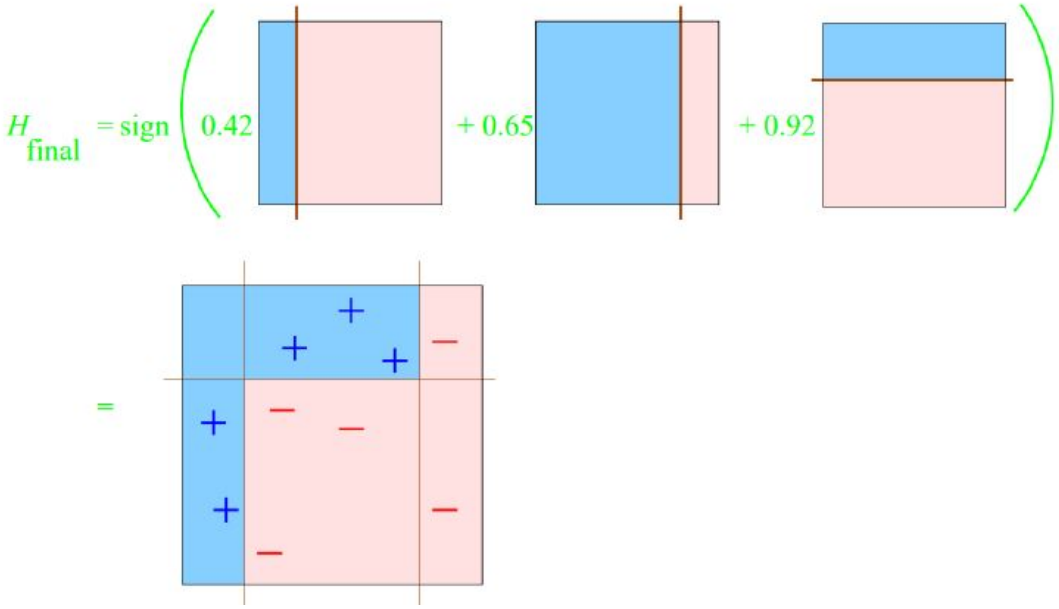
$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i),\ f_k \in \mathcal{F}$$
$$where \quad \mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}\ (q : \mathbb{R} \to T,\ w \in \mathbb{R}^T)$$

- $\mathcal{F}$: 트리 모델의 공간 (CART), T : the number of leaves,

  q : index로 유도하는 개별 트리, $w_{q(\mathbf{x})}$: leaf의 개별적 트리에 대한 가중치

  - **Boosting의 과정**

  1. 목표값을 찾기 위해 f_1을 적용해 데이터셋을 분할
  2. f_1에서 오분류된 obs들에 가중치를 부여해서 f_2에서 더 자주 선택되도록 유도
  3. f_2를 통해 재분류를 진행, 오분류된 obs들에 가중치 부여로 자주 선택되도록 유도
  4. 위와 같은 과정을 반복해 앞의 모델에서 설명하지 못했던 정보를 이후 모델이 보완하는 식으로 지속적인 학습을 진행하여 복잡한 Boundary 형성



출처 : Ensemble Learning: Adaptive Boosting (AdaBoost), Pilsung Kang, School of Industrial Management Engineering, Korea University

- 해당 과정을 통해 트리 앙상블 모델을 통한 최종 예측값은 **개별 트리의 예측값 w_i의 합**으로 구할 수 있게 된다.
- 트리 모델의 성능을 높이기 위해서는 정규화된 목적함수를 **최소화**하는 과정이 필요하다. 목적함수는 다음과 같이 표시할 수 있다.

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) \ + \ \sum_k \Omega(f_k)$$
$$where\ \Omega(f) = \gamma T \ + \frac{1}{2}\lambda \|w\|^2$$

- $\mathcal{L}(\phi)$: 목적함수, $l(\hat{y}_i, y_i)$: 목표값과 예측값의 차이를 나타내는 convex한 **loss function**
- $\Omega(f)$ = 모델을 smooth하게 만들어주는 **penalize function** (과적합 방지)
- 위의 목적함수는 Regularized greedy forest(RGF)에서 사용된 목적함수와 유사하나, 더 간단하고 병렬화가 더 쉽다는 장점이 존재한다.
- $\Omega = 0$일 경우 전통적인 gradient tree boosting과 동일한 목적함수가 된다.

2.2 Gradient Tree Boosting

- 앞에서 제시된 트리 앙상블 모델은 목적함수 내에 함수를 포함하고 있으므로, 유클리드 공간에서 최적화될 수 없다.
  - 유클리드 공간에서 최적화가 되려면 parameter가 스칼라 형태로 존재해야한다.
- 따라서 모델은 Additive한 방식으로 학습이 진행된다.
- $\hat{y}_i^{(t)}$를 t번째 분할을 했을 때의 i번째 instance라고 할 때, 목적함수를 최소화하기 위해서 f_t를 추가해주어야 한다.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} \ + \ f_t(\mathbf{x}_i)) \ + \ \Omega(f_t)$$

- 이해가 쉽게 해당 식을 풀어서 설명하면 다음과 같다.

$$\mathcal{L}^{(t)} = f_1 \ + \ f_2 \ + \ \cdots \ + f_{t-1} \ + f_t \ + \ \Omega(f_t)$$

- f_1부터 f_{t-1}까지는 t-1번 분할했을 때까지의 loss function의 합을 의미하며, $l(y_i, \hat{y}_i^{(t-1)})$로 표현될 수 있다.
- f_t를 통해서 t번째 분할을 진행함으로써 t-1번까지에서 설명이 되지 않았던 데이터를 설명할 수 있게 된다.
- 따라서 우리는 greedy하게 f_t를 추가함으로써 모델 성능을 가장 향상시킬 수 있다.
- 해당 목적함수의 빠른 최적화를 위해 일반적으로 테일러 이차근사가 사용된다.
  - **참고 : 테일러 다항식**

    - 함수는 테일러 다항식을 통해 표현될 수 있고, 이를 이용해 함수의 근삿값을 구할 수 있다.

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!} + R_n(x)$$

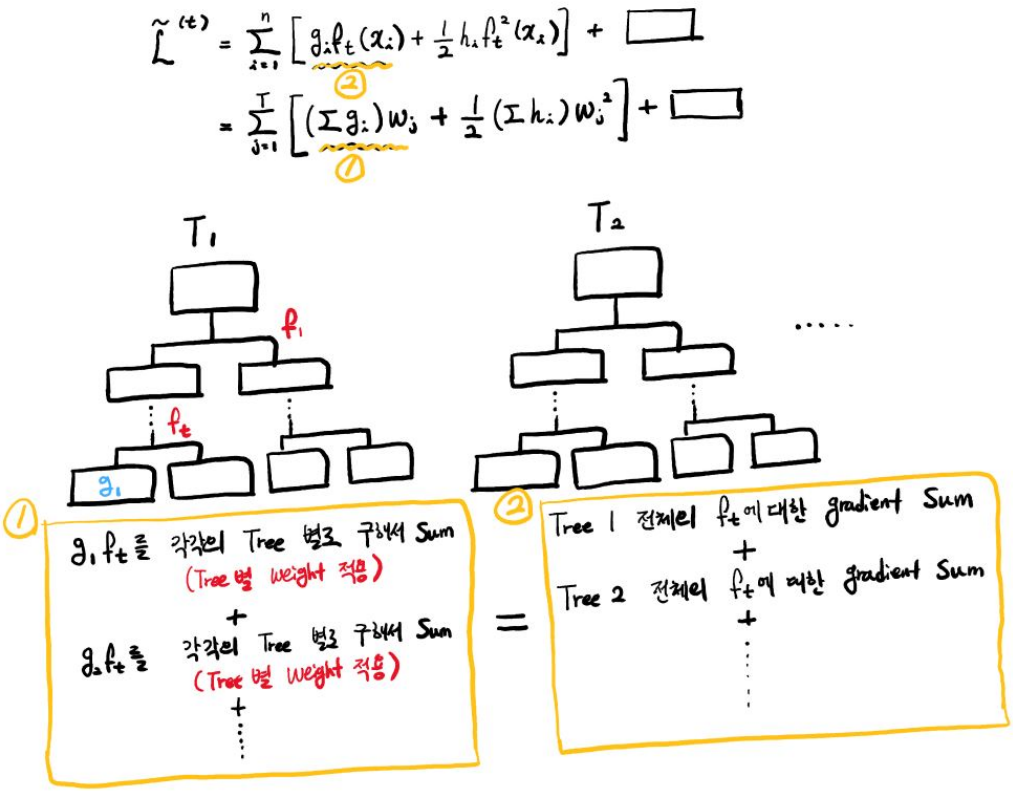
- 이를 통해 목적함수를 근사하면 다음과 같이 표현할 수 있다.

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) \; + \; g_i f_t(\mathbf{x}_i) \; + \; \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] \; + \; \Omega(f_t)$$
$$where \quad g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$$

- 목적함수를 t에 대한 함수의 관점에서 보면, $\sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}]$은 상수항으로 취급된다.
- g_i와 h_i는 각각 1차미분과 2차미분을 간단하게 표현하기 위하여 사용되었다.
- 우리는 t번째 분할에 대해서 최소화를 시키는 것이 목적이기 때문에 앞의 상수항 부분은 제거한다. 또한 $\Omega(f_t)$도 풀어서 전개해주면 다음과 같은 식을 얻을 수 있다:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) \; + \; \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] \; + \gamma T \; + \; \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

- 다음 식을 간단하게 만들기 위해서는 $\sum_{i=1}^{n}$와 $\sum_{j=1}^{T}$, 두 합을 하나의 관점으로 묶어줘야 하고, 개념적으로 이를 묶을 수 있다.
- 이때, Loss function은 ϕ에 관한 함수이고, ϕ는 Tree Boosting에 관한 모델이기 때문에 Tree의 관점에서 $\sum$을 바라봐야 한다. 따라서 T로 해당 합을 묶어준다.
    - **참고 : w_j에 관한 식으로 변환되는 과정**



- $g_i f_t(x_i)$가 $(\sum g_i)w_j$로 변환되는 이유는 **1) 트리별로 g_i를 구해서 트리별 가중치를 곱해 모든 i에 대해 합하는 과정**과 **2) 트리 전체의 g를 구해서 트리를 모두 더하는 과정**이 같기 때문이다. 해당 경우는 h_i에도 동일하게 적용된다.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)w_j \ + \ \frac{1}{2}(\sum_{i\in I_j} h_i \ + \ \lambda)w_j^2] \ + \ \gamma T$$

- $I_j = \{i \mid q(\mathbf{x}_i) = j\}$: 트리 q에 x_i를 넣으면 j index에 배정
- 목적함수에 관한 수식을 정리하면 위와 같이 쓸 수 있다.
- 이제 우리의 최종 목적인 목적함수 $\mathcal{L}$을 최소화하기 위해서 가중치인 w_j의 최적값을 찾아주는 과정만이 남았다.
- w_j의 최적값을 찾기 위해 $\mathcal{L}$을 w_j에 관하여 편미분하면, 1차 미분이 0인 지점이 최적의 w_j이다.

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i \ + \ \lambda}$$

$$\tilde{\mathcal{L}^{(t)}}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i \ + \ \lambda} \ + \ \gamma T$$

- w_j의 최적값은 위와 같고, 해당 최적 가중치를 목적함수에 대입하면 그 다음과 같이 트리 q의 성능을 평가하기 위한 평가 함수를 만들어낼 수 있다.
- 해당 평가 함수는 더 넓은 목적함수의 범위에서 구해진다는 점을 제외하면 의사결정 나무의 불순도 평가와 유사하다.

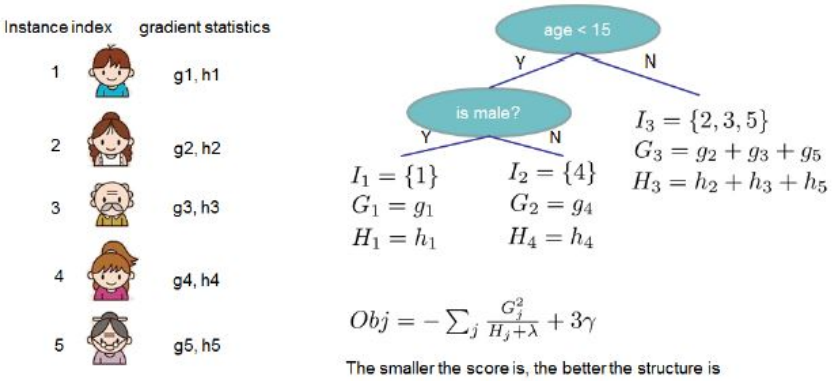


Figure 2: Structure Score Calculation. We only need to sum up the gradient and second order gradient statistics on each leaf, then apply the scoring formula to get the quality score.

최적화된 목적함수를 통해서 Quality score를 평가하는 예시.

- 일반적으로 가능한 모든 트리 q에 대해서 나열하는 것은 불가능하다. 따라서 그 대안으로 작은 leaf에서 시작해서 반복적으로 가지를 더해나가는 greedy algorithm이 사용된다.
- I_L을 분리 이후 좌측에 배정되는 instance set, I_R을 분리 이후 우측에 배정되는 instance set이라고 가정하고, $I = I_L \cup I_R$이라고 두면 분리 이후 loss reduction은 다음과 같이 구해진다.

$$\mathcal{L}_{split} = \frac{1}{2}[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i \ + \ \lambda} \ + \ \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i \ + \ \lambda} \ - \ \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i \ + \ \lambda}] \ - \ \gamma$$

- 해당 지표를 통해 split candidate를 일반적으로 평가할 수 있다.
  - 분리 이후 왼쪽 node에 대한 성능과 오른쪽 node에 대한 성능을 구한 뒤, 분리 전 성능을 빼주는 형태를 띄고 있음을 알 수 있다.

2.3 Shrinkage and Column Subsampling

- 위에서 소개된 Tree Boosting 방법들은 모델을 예측하고 분류하는 데 일반적으로 좋은 성능을 보유하고 있다. 하지만 Tree Boosting 계열의 방법들은 과적합이 발생할 수 있다는 문제 또한 자명한 사실이다.
- 과적합을 방지하기 위해서 해당 논문에서는 2가지 방법이 소개되었는데, **Shrinkage**와 **Column Subsampling**이다.

1. **Shrinkage**
    - Shrinkage는 개별 트리의 영향력을 줄이는 방법으로, 적은 수의 large step보다 많은 수의 small step을 택함으로써 모델의 성능을 향상시킨다.
    - 초기 split 이후 지속적으로 **가중치** η를 곱해주면서 추후 모델의 영향력을 지속적으로 감소시키며, 후속 모델들을 위한 공간을 확보한다.

2. **Column Subsampling**
    - 학습을 반복하는 과정에서, traing set의 랜덤한 일부분만 학습에 사용하며 과적합을 방지하는 방법이다.
    - **RandomForest** 모델에서도 해당 방식을 적용하여 과적합을 방지하고 있다.
    - 전통적인 row sub-sampling 과정보다 더 많은 과적합을 방지할 수 있다는 유저들의 피드백이 존재한다.
    - 보편적으로는 비복원 추출이 train sampling에 사용되나, bagging으로 대표되는 복원추출 방법도 사용이 가능한 것으로 알려져있다.
    - 데이터의 일부를 추출해 f_1을 만들고, 이를 통해 $f_2(x) = y - f_1(x)$와 같은 과정을 거쳐 다시 Original Dataset에 넣은 후 새로 sampling을 하는 방식으로 진행된다.
        - 해당 방식으로 진행되기 때문에 데이터가 지속적으로 줄어드는 문제가 발생하지 않는다.
        - 추후 언급될 병렬적 알고리즘의 계산 속도 증가에도 도움을 준다.

> 출처 : Ensemble Learning: Gradient Boosting Machine (GBM), Pilsung Kang, School of Industrial Management Engineering, Korea University

3. Split Finding Algorithms

> 그림 참고: *고려대학교 DSBA 연구실 강의*

3.1 Basic Exact Greedy Algorithm

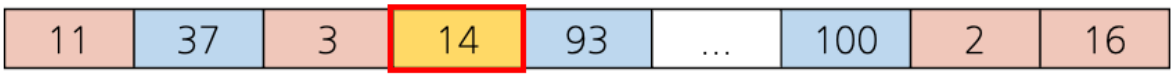
Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted*$(I,$ *by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

논문에 제시된 Basic Exact Greedy Algorithm 설명

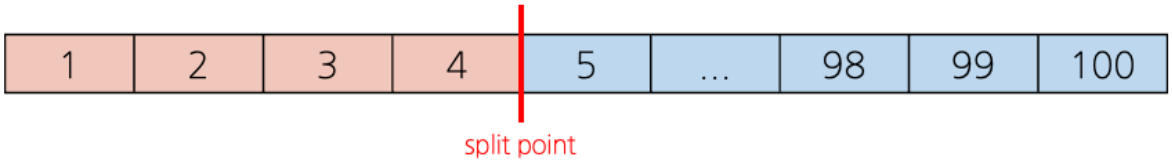
Basic exact greedy algorithm은 모든 feature에 대하여 모든 가능한 split을 탐색하는 방법이다. 그런데, 값의 크기에 관계 없이 뒤죽박죽 섞인 데이터, 즉 unsorted 데이터를 이용해서 무작정 탐색을 진행하게 된다면 특정 split point가 주어졌을 때 모든 instance에 대해 그 값이 큰지, 작은지 하나하나 비교를 해 주어야 하므로 계산 비용이 매우 많이 든다는 문제가 발생하게 된다. 특히, 최적의 split을 찾기 위해서는 모든 instance를 기준으로 분할을 진행한 후 각각의 score 값을 비교해야 한다는 사실을 떠올려 보면 계산 과정을 조금 더 효율적으로 바꿀 필요가 있음을 알 수 있다. 이 문제를 해결하기 위해 basic exact greedy algorithm에서는 feature의 값에 따라 데이터에 대한 sorting을 진행하게 된다.

Sorted 데이터를 이용했을 때의 장점을 예시를 들어 설명해보도록 하겠다. 우선, 1부터 100까지의 자연수를 나열하는 상황을 가정해 보자. 그리고 아래와 같이 자연수를 나열했다고 생각해 보겠다.



위의 숫자들은 순서가 뒤죽박죽 섞인, unsorted 데이터임을 쉽게 알 수 있다. 이 상태에서 split point에 대한 탐색을 진행한다고 하면, 예를 들어 14를 기준으로 잡았다고 하면 11, 37, 3, 93, 100, 2, 16 그리고 나머지 자연수들 모두에 대해 그 값이 14보다 큰지 작은지를 판단을 해 주어야 한다. 마찬가지로 이 과정을 11, 37, 3, 93, ..., 100, 2, 16 나머지 값 각각을 split point로 잡은 후 값을 비교하는 과정을 계속 반복해 주어야 한다.

그렇다면, 이번에는 1부터 100까지의 자연수를 오름차순으로 정렬하여 다음과 같이 나타냈다고 가정해 보자.



위의 숫자들은 처음의 예시와 달리 크기에 따라 순서대로 정렬된 sorted 데이터임을 알 수 있다. 이 상태에서 split point에 대한 탐색을 진행한다고 하면, 예를 들어 4와 5 사이를 split point로 잡았다고 하면 자연수들이 크기 순서대로 정렬이 되어 있기 때문에 split point를 기준으로 좌측의 값은 모두 작고, 우측의 값은 모두 크다고 판단할 수 있다. 모든 instance에 대한 크기를 비교하는 과정 없이 split point를 기준으로 좌/우만 나누면 된다는 점에서 계산량이 크게 줄게 되고, 따라서 gain(score)을 구하는 과정을 빠르게 처리할 수 있다.

Basic exact greedy algorithm은 모든 feature에 대하여 모든 가능한 split을 탐색하기 때문에 항상 최적해(optimal split)를 보장한다는 장점을 지닌다. 그러나, 전체 데이터를 이용하여 탐색을 진행하므로 데이터 전체가 한꺼번에 메모리에 로드되지 않으면 탐색 자체가 불가능하다. 또한, 같은 특성으로 인하여 분산 환경에서는 처리가 불가능하다는 단점을 가진다.

3.2 Approximate Algorithm

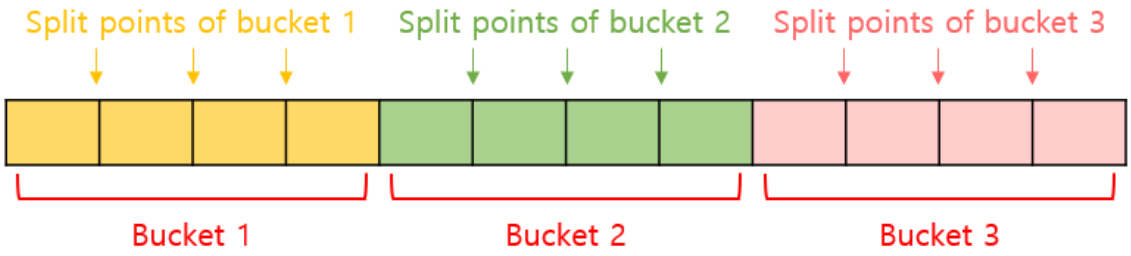
Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ ***to*** m **do**
  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.
  Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ ***to*** m **do**
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max score only among proposed splits.

논문에 제시된 Approximate Algorithm 설명

Basic exact greedy algorithm과 비교되는 approximate algorithm의 가장 큰 특징은 바로 candidate split points를 제시한다는 것이다. 이 candidate split points는 특정 feature에 대한 분포의 percentile에 따라 제시된다. 즉, 각 feature에 대하여 데이터를 정렬하고, 정렬된 분포에서 percentile에 따라 데이터를 분할하게 된다. 데이터가 분할됨에 따라 각 split point 사이, 그리고 맨 앞과 맨 뒤에 데이터의 묶음(또는 그룹)이 생기게 되는데, 이 묶음을 bucket(또는 block)이라고 한다.

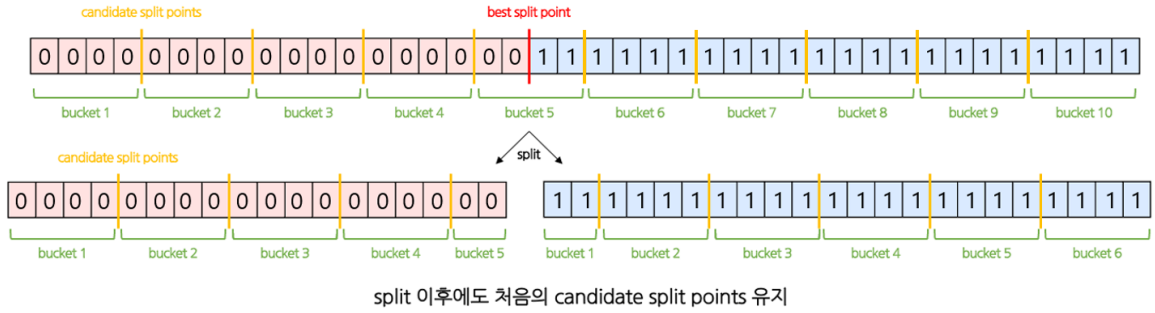
Percentile에 따라 bucket들이 정해진 후에는 각각의 bucket안의 모든 split points에 대하여 g_i, h_i를 구한다. 이후, 각 bucket별로 g_i, h_i를 합산하여 가장 큰 score 값을 갖는 bucket을 기준으로 분할을 진행한다. 그림으로 나타내면 아래와 같다.



Approximate algorithm의 장점은 각 bucket에 대해 통계량(gradient)을 구하기 때문에 병렬 처리가 가능하고, basic exact greedy algorithm에 비해 빠른 계산이 가능하다는 것이다. 그러나, 최적의 split point를 보장하는 basic exact greedy algorithm과는 달리 approximate algorithm은 최적의 split point를 보장하지는 못한다.

Approximate algorithm에는 global variant, local variant라고 불리는 두 가지의 variants가 있는데, 이는 candidate split points가 제시되는 시점에 따라 구분된다. 간략히 정리하자면, global variant는 per tree이고, local variant는 per split이다.

우선, global variant는 트리를 처음 생성할 때의 split 기준을 그대로 사용한다. 따라서, 트리가 깊어짐에 따라 candidate split points가 변화하지 않으며, 모든 level에 대하여 같은 candidate split points를 사용한다. 그림으로 나타내면 아래와 같다.

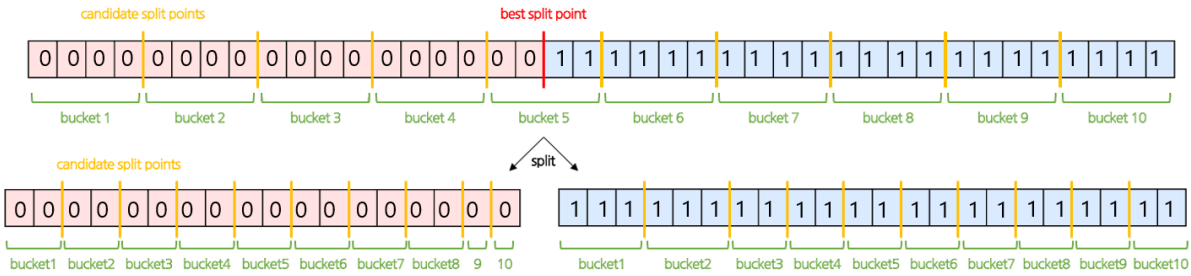


split 이후에도 처음의 candidate split points 유지

global variant의 특징

이처럼, global variant에서는 트리가 깊어질수록 bucket size가 유지되지 않고 오히려 감소하게 된다. 이러한 특징에 따라 global variant는 proposal 단계가 적기는 하지만 global proposal시 더 많은 candidate split points가 필요하다는 특징을 가진다.

다음으로, local variant는 각 split 이후 candidate split points를 다시 제시한다. 예를 들어, bucket size를 10으로 설정했다고 하면, split 이후 생기는 좌, 우 각각의 child 노드 내부에서 percentile에 따라 다시 10개의 bucket을 제시한다. 이를 그림으로 나타내면 아래와 같다.



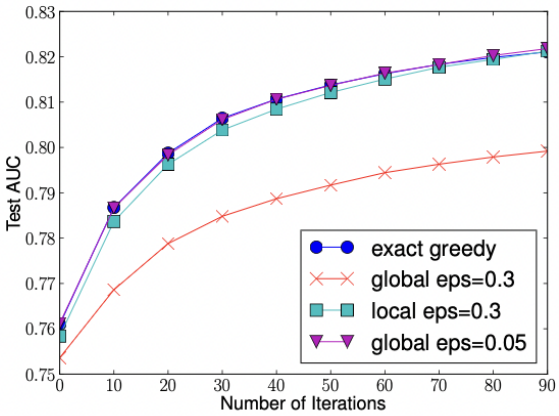
split 이후 새로운 candidate split points 제시, 이 때 bucket의 수는 유지

local variant의 특징

이처럼, local variant에서는 각각의 child 노드에 대해 percentile을 다시 계산해서 bucket size를 일정하게 유지한다. 이로 인하여 트리가 깊어질수록 하나의 bucket 안에 있는 example의 수가 감소하게 된다.

여기에서, bucket을 나누는 기준인 candidate split points의 개수와 연관된, percentile을 얼마나 잘게 나눌지에 대한 파라미터인 ϵ(epsilon)이 등장한다. 이 때, ϵ의 역수인 $1/\epsilon$은 생성되는 bucket의 수를 나타낸다.

앞에서 언급한 바와 같이, global variant는 트리가 깊어질수록 bucket size가 감소하기 때문에, 초기의 global proposal 단계에서 많은 수의 candidate split points가 제시되어야 한다. 따라서, global variant 사용 시에는 ϵ을 작게 잡아야 한다.



위의 그림은 basic exact greedy algorithm, global variant을 이용한 approximate algorithm, local variant을 이용한 approximate algorithm의 성능을 비교한 것이다. global variant만 우선 보자면, global variant에서는 ϵ의 값이 작을수록, 즉 bucket의 수가 많을수록 성능이 좋은 것을 확인할 수 있다. 또한, 위의 그림을 통해 approximate algorithm에서 ϵ값만 잘 조정해 준다면 basic exact greedy algorithm 못지않은 성능을 낼 수 있다는 사실을 확인할 수 있다.

3.3 Weighted Quantile Sketch

특정 feature에 대한 데이터를 아래와 같이 나타내어보자.

$$\mathcal{D}_k = \{(x_{1k}, h_1),\ (x_{2k}, h_2),\ \ldots,\ (x_{nk}, h_n)\}$$

여기에서 x_{ik}는 k번째 feature의 i번째 instance값을 나타내고, h_i는 i번째 instance에서의 2차 미분 값을 나타낸다.

$\mathcal{D}_k = \{(x_{1k}, h_1),\ (x_{2k}, h_2),\ \ldots,\ (x_{nk}, h_n)\}$에서 각 datapoint는 h_i만큼의 가중치를 가진다. 2차 미분값이 각 datapoint의 가중치가 되는 이유는 다음과 같다. 앞서 근사한 regularized objective를 다시 떠올려보면,

$$\begin{aligned}
\tilde{L}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \\
&= \sum_{i=1}^{n} \frac{1}{2} h_i [f^2(x_i) + \frac{g_i}{h_i} f_t(x_i)] + \Omega(f_t) \\
&= \sum_{i=1}^{n} \frac{1}{2} h_i [f_t(x_i) + \frac{g_i}{h_i}]^2 + \Omega(f_t) + constant
\end{aligned}$$

위와 같은 형태로 식을 정리할 수 있다. 따라서, 이 식을 weighted squared loss로 볼 수 있다.

일반적으로, 특정 feature의 percentile은 정렬된 데이터 상에서 candidate split points가 균등하게 분포되도록 한다. 그러나, 위와 같이 우리가 가진 데이터는 모든 데이터가 동일한 취급을 받는 것이 아닌, 서로 다른 가중치가 부여된 데이터이다. 따라서, 우리가 가진 데이터의 경우 데이터를 정렬한 후 단순 percentile대로(일정한 간격대로) 잘라서 split points를 찾는 것이 아니라 각 데이터의 weight를 반영해서 split points를 찾아 주어야 한다.

그러나, 현재는 가중치가 부여된 데이터셋에 대한 quantile sketch가 존재하지 않아 현존하는 근사 알고리즘의 대다수는 정확하지 않을 가능성이 존재한다. 따라서, XGBoost는 rank function을 이용한 weighted quantile sketch를 수행하여 candidate split points를 찾는 데 증명 가능한 이론적 근거를 바탕으로 한 알고리즘을 제시하고자 한다.

Rank function $r_k = \mathbb{R} \to [0, +\infty)$는 다음과 같이 정의할 수 있다.

$$for\ the\ instances\ in\ feature\ k, \quad r_k(z) = \frac{1}{\sum_{(x,\ h) \in \mathcal{D}_k} h} \sum_{(x,\ h) \in \mathcal{D}_k .\ x < z} h$$

이는 feature k에서 z보다 작은 값을 가진 instance들의 2차 미분값의 합을 전체 instance에 대한 2차 미분값들의 합으로 나누어 줌으로써 fearure k에 대한 값이 z보다 작은 instance들의 비율을 나타낸다.

위의 rank function은 candidate split points인 $\{s_{k1}, s_{k2}, \ldots, s_{kl}\}$을 찾는 것을 목표로 한다. 이 때, $\{s_{k1}, s_{k2}, \ldots, s_{kl}\}$는 아래와 같은 성질을 만족한다.

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon, \quad s_{k1} = \min_i \mathbf{x}_{ik}, s_{kl} = \max_i \mathbf{x}_{ik}$$

여기에서 ϵ은 percentile을 얼마나 잘게 쪼갤지 결정하는 approximation factor를 나타낸다.

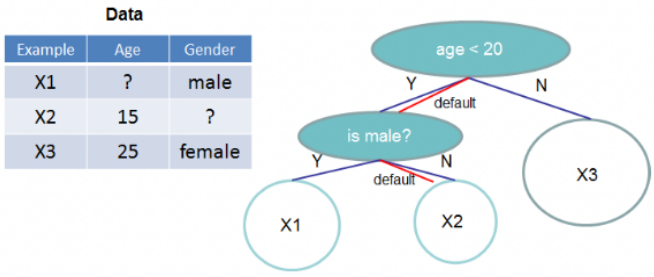
특히, XGBoost는 merge와 prune operation을 적용할 수 있는 데이터의 구조를 제안한다. 메모리에는 한계가 존재하기 때문에 데이터를 쪼개서 각각에 대한 quantile summary를 산출해야 하고, 이후 따로 작업한 결과들을 merge 해야 한다. 더불어, 최적의 candidate split points로 element들을 줄이기 위해서는 pruning이 필요하다.

3.4 Sparsity-aware Split Finding

데이터가 빈 부분 없이 빽빽하게 모두 채워져 있다면 좋겠지만, 이는 이상일 뿐이다. 우리가 가진 입력 데이터의 대다수는 sparse하다. Sparse data의 대표적인 원인 세가지는 다음과 같다.

- 데이터에 결측치가 존재
- 통계량에서 zero entries의 빈번한 등장
- 원핫인코딩(one-hot encoding)과 같은 feature engineering

이러한 상황에서는 알고리즘이 데이터의 sparsity를 알아야 한다. 이는 트리의 각 노드에 default direction을 설정함으로써 해결할 수 있다. 즉, sparse matrix X에서 값이 누락되어 있는 경우에는 split 시 instance가 default direction으로 이동한다.



위의 그림에 제시된 데이터셋에서는 첫 번째 example의 Age 값이 누락되어 있다. 따라서, 첫 번째 split에서는 default direction인 왼쪽으로 분류된다. 그 후, 첫 번째 example의 Gender 값이 male이기 때문에 두 번째 split에서는 조건에 맞게 왼쪽으로 분류된다. 두 번째 example의 경우에는 Age 값이 존재하기 때문에 우선 첫 번째 split에서는 조건에 맞게 왼쪽으로 이동한다. 그러나, 두 번째 split에서는 Gender 값이 누락되어 있기 때문에 default direction인 오른쪽으로 이동하게 된다. 이처럼 누락된 값이 존재할 때 default direction으로 분류하는 방법이 바로 sparsity-aware split finding이다.

최적의 default direction은 데이터로부터 학습된다. 그 과정은 아래와 같이 정리해 볼 수 있다.

Algorithm 3: Sparsity-aware Split Finding

Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I}, g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  // enumerate missing value goto right
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted*$(I_k,$ *ascent order by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
  // enumerate missing value goto left
  $G_R \leftarrow 0,\ H_R \leftarrow 0$
  for j *in sorted*$(I_k,$ *descent order by* $\mathbf{x}_{jk})$ **do**
    $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
    $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default directions with max gain

논문에 제시된 Sparsity-aware Split Finding 설명

1. 모든 결측치를 오른쪽 노드로 이동시킨 후 non-missing 데이터에 대하여 모든 split point candidates를 탐색한다.
2. 모든 결측치를 왼쪽 노드로 이동시킨 후 non-missing 데이터에 대하여 모든 split point candidates를 탐색한다.
3. 모든 결측치를 오른쪽 노드로 이동시킨 경우와 모든 결측치를 왼쪽 노드로 이동시킨 경우의 gain(score)을 비교하여 최적의 default direction을 결정한다.

4. SYSTEM DESIGN

XGBoost는 분산 처리 환경에서 학습 가능하고 CPU 캐시를 고려한 알고리즘이라 데이터의 크기가 커져도 빠른 속도로 학습할 수 있다는 점을 설명하고 있다.

4.1 Column Block for Parallel Learning

data를 sort하는건 time consuming 하기에 이에 드는 비용을 절감하기 위해서 block이라고 부르는 in-memory unit에 데이터를 저장한다.

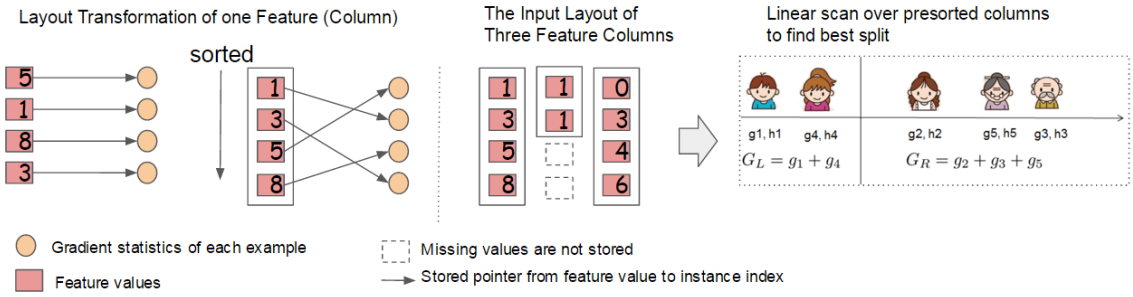
각 블락에서 데이터는 CSC(compressed column)형태로 저장된다.

- **csc**

  각 column이 대응하는 feature value에 따라 정렬된 형태로 이러한 데이터 구조는 training전에 한번 계산되면 이후 반복에서 재사용 가능하다.

- **greedy algorithm**

  한 개의 block에 모든 데이터셋을 저장하고 split search algorithm을 적용하기 위해 linear scan을 실시한다. → column별로 best split을 빠르게 찾기 위한 linear scan을 시행한다.



- **approximate algorithm**

  앞선 greedy algoritm과 달리 multi block을 사용한다는 것이 특징이다. 각 block은 row의 subset들이다. 각 block에서 best split을 구하고 이후에 그 best split에 대한 linear scan을 진행할 수 있다.

  → 서로 다른 블록들은 여러 기계들에 분산되거나 out-of-core환경의 디스크에 저장될 수 있다.

  각 branch에서 후보가 자주 생성되는 local proposal에서 가치가 있다.

- **parallelized**

  각 column에 대한 통계량을 수집하는 것은 병렬화 될 수 있다. 이것은 split finding에 대한 병렬 알고리즘을 제공한다.

4.2 Cache-aware Access

- **cache**

  자주 사용하는 데이터나 값을 미리 복사해 놓는 임시 장소로 저장 공간이 작고 비용이 비싼 대신 빠른 성능 제공한다.

  반복적으로 데이터를 불러오는 경우에 지속적으로 서버에 요청하는 것이 아니라 memory에 데이터를 저장하였다 불러 쓰는 것을 의미한다.

제시된 block 구조가 split finding의 계산 복잡성을 최적화하는데 도움을 주지만, 새로운 알고리즘은 이 값을 feature 순서에 따라 접근하기 때문에 row 인덱스에 따라 gradient statistics의 간접적인 fetches를 필요로 한다.

- **fetch 매서드**

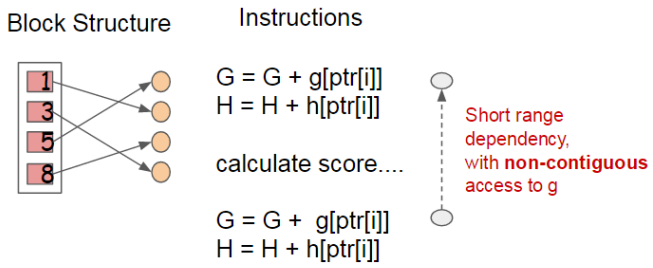
  자바스크립트에서 서버로 네트워크 요청을 보내고 응답을 받을 수 있도록 해주는 메서드

  api를 불러오고 내보내주는 함수

→ 이것은 불연속적(non-continuous) 메모리 접근방식이다.

불연속적 메모리 할당 : 메모리 공간의 다른 위치에 별도의 메모리 블록을 비 연속적으로 할당하는 것이다.

연속 메모리에 비해 프로세스가 비교적 느리게 시행된다.



→ naive한 분할 열거의 시행은 누적과 불연속적 메모리 fetch 작업 사이의 즉각적인 read/write 종속성을 도입한다.

→ gradient statistics가 cpu 캐시에 적합되지 않고 캐시 미스가 발생할 때, 분할 탐지 알고리즘을 느리게 한다.

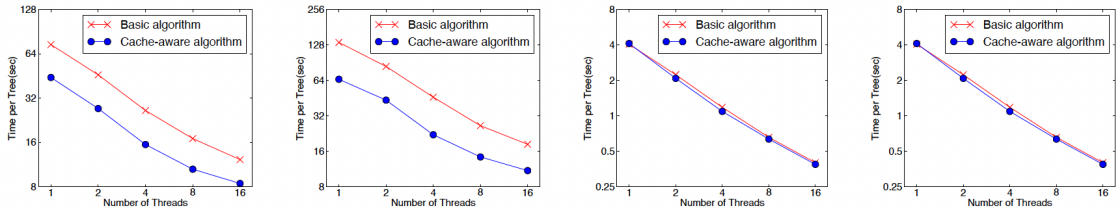
캐시 미스 : cpu가 데이터를 요청하여 캐시 메모리에 접근했을 때 캐시 메모리에 해당 데이터가 없어서 DRAM에서 가져와야 하는 경우

- **Exact greedy algorithm**

  Exact greedy algorithm의 경우, *cache-aware prefetching algorithm*을 사용하여 이러한 문제를 완화할 수 있다.

  data prefetch : 연산에 필요한 data들을 미리 가져오는 것이다.

  이 prefetching 방식은 직접적인 read/write 종속성을 더 긴 종속성으로 변경하고, row수가 클 때 overhead시간을 감소시킨다.



(a) Allstate 10M (b) Higgs 10M (c) Allstate 1M (d) Higgs 1M

- **Approximate algorithm**

Appriximate algorithm의 경우에는 이 문제를 정확한 블록 사이즈를 선택함으로서 해결한다. 블록사이즈를 브록에 포함된 예시의 최대 수로 정의한다. 너무 작은 블록 사이즈는 각 thread에서 작아져 비효율적인 병렬화를 유도한다. 반면 너무 큰 블록의 크기는 gradient 통계량이 CPU캐시에 적합되지 않기 때문에, 캐시 미스를 야기한다.

thread : 프로그램(프로세스) 실행의 단위

4.3 Blocks for out-of-core Computation

앞서 소개한 cache만을 활용하면 좋겠지만 scalable machine learning을 달성하기 위해선 machine을 최대로 효율적으로 사용하는 것이 좋은데, 이때 cache만 사용하는 것 보다 disc도 함께 활용해 메인 메모리에 적합되지 않은 데이터를 다루는 것이 필요하다.

- **out-of-core**

  컴퓨터의 메인 메모리에는 너무 큰 데이터를 가공하는 것을 지칭

  out-of-core 계산 수행을 위해 데이터를 다중 블록에 나누고, 각 블록을 디스크에 저장한다. 그러나 이 과정에서 디스크 reading이 계산 시간 대부분을 차지하기 때문에 이러한 오버헤드를 감소시키는 것이 중요하다. 아래의 두 방법을 활용할 수 있다.

1. block compression
2. block sharding

5. Conclusion

이 논문에서 데이터 사이언티스트들에 의해 광범위하게 활용되며 여러 문제에 대한 좋은 결과를 제공하는 확장가능한 tree boosting system인 XGBoost에 대해 배웠다. 여러 알고리즘과 패턴등은 다른 머신러닝 시스템에 또한 적용가능하다. 위 논문에 제시된 인사이트들을 결합함으로서 XGBoost는 최소한의 자원만을 사용하여 실제의 광범위한 문제를 해결할 수 있다.